Techniques for Fast and Scalable Time Series Traffic Generation

Jilong Kuang, Daniel G. Waddington and Changhui Lin
Computing Science Innovation Center
Samsung Research America
Email: {jilong.kuang, d.waddington, changhui.lin}@samsung.com

***Abstract*—Many IoT applications ingest and process time series data with emphasis on 5Vs (Volume, Velocity, Variety, Value and Veracity). To design and test such systems, it is desirable to have a high-performance traffic generator specifically designed for time series data, preferably using archived data to create a truly realistic workload. However, most existing traffic generator tools either are designed for generic network applications, or only produce synthetic data based on certain time series models. In addition, few have raised their performance bar to millions-packets-per-second level with minimum time violations.**

In this paper, we design, implement and evaluate a highly efficient and scalable time series traffic generator for IoT applications. Our traffic generator stands out in the following four aspects: 1) it generates time-conforming packets based on high-fidelity reproduction of archived time series data; 2) it leverages an open-source Linux Exokernel middleware and a customized userspace network subsystem; 3) it includes a scalable 10G network card driver and uses "absolute" zero-copy in stack processing; and 4) it has an efficient and scalable application-level software architecture and threading model. We have conducted extensive experiments on both a quad-core Intel workstation and a 20-core Intel server equipped with Intel X540 10G network cards and Samsung's NVMe SSDs. Compared with a stock Linux baseline and a traditional mmap-based file I/O approach, we observe that our traffic generator significantly outperforms other alternatives in terms of throughput (10X), scalability (3.6X) and time violations (46.2X).

I. Introduction

The Internet-of-Things (IoT) is emerging as the third wave in the development of the Internet [25]. The next two decades will see many technology advances driven by IoT. Gartner forecasts that 25 billion connected things will be in use by 2020 [3]. Business, finance, retail, manufacturing, utilities, health, education, transportation, agriculture, mining and every other sector will be directly impacted by IoT [29].

Many IoT applications ingest time series data (*e.g.*, from sensors, smart devices, wearables) and process them with emphasis on 5Vs (Volume, Velocity, Variety, Value and Veracity). To facilitate testing and optimizing such complex applications/systems, it is desirable to have a high-performance traffic generator specifically designed for time series data, preferably using archived data to create a truly realistic workload. Undoubtedly, such a traffic generator will be useful in many areas. For example, application developers can use it to test and analyze their systems; data scientists can carry out data analytics in a real-time fashion; system admins and cloud providers will be able to accurately estimate traffic load and design fine-grained resource management and provisioning strategies.

However, most of the existing works have focused on building realistic models to generate synthetic traffic for various network environments (*e.g.*, [12], [13], [15], [34], [36] and [37]). Their objective is to simulate the traffic pattern as close as possible to the real world. Although promising to some extent, these systems suffer from the following issues with respect to IoT applications: 1) they are unable to capture the "true" characteristics of dynamic live streaming of time series data; no matter how accurate the model is, it will not be as accurate as actual data traffic; 2) they fall short of satisfying IoT applications' all 5Vs requirement due to suboptimal performance in terms of throughput, scalability and time violation, as optimizing those metrics are not their primary goals; 3) they are not time-series-data-centric, thus lack simulation flexibility such as probabilistic out-of-order simulation and drifted time window simulation.

In this paper, we design, implement and evaluate a highly efficient and scalable time series traffic generator for IoT applications. Our goal is to generate time-conforming packets based on high-fidelity reproduction of archived time series data. Under such a requirement, we aim for *maximum throughput*, *better scalability* and *minimum time violations*. In addition to superior performance, we also aim to add various simulation features into the traffic generator, including probabilistic out-of-order simulation, multi-flow simulation, records with varying interval simulation and drifted time window simulation.

There are a number of challenges in developing such a traffic generator. The novel approach in our design is to adopt an aggressive whole-system optimization, ranging from OS and network subsystem (userspace driver and stack) to application design and threading model. More specifically, we leverage an open-source Linux Exokernel middleware [2] (also developed by us) to customize OS core functionalities so that we can bypass the slow Linux kernel and improve scalability in memory management and interrupt handling. On top of that, we develop a network subsystem that is completely sitting in user space. Our customized 10G Network Interface Card (NIC) driver and zero-copy technique in stack processing guarantee efficient and scalable packet transmission. In addition, we design a multi-queue software architecture along with a double-buffer scheme for traffic generator application.

We have conducted extensive experiments to evaluate the performance of our traffic generator on both a quad-core Intel workstation and a 20-core Intel server. We have leveraged Intel X540 10G cards and Samsung's NVMe SSD in our platform

for best performance. Not only have we tested our system using various synthetic data traces, but also we have run a real 24-hour seismic data [11] (about 2.8 billion time series sensor records in a single file). Compared with a stock Linux baseline and a traditional mmap-based file I/O approach, we observe that our traffic generator produces superior performance and outperforms other alternatives significantly in terms of throughput (10X), scalability (3.6X) and time violations (46.2X). To summarize, we make the following contributions:

- We propose a highly efficient and scalable time series traffic generator on a multi-core architecture. To the best of our knowledge, this is the first work towards millions-packets-per-second time-conforming traffic generation based on high-fidelity reproduction of archived time series data.
- We develop a userspace network subsystem via aggressive customization of the OS core functionalities based on an open-source Linux Exokernel middleware.
- We apply an "absolute" zero-copy technique to eliminate any memory copy at all levels. Packet transmission is handled very efficiently to achieve the best throughput and latency.
- We design an efficient and scalable software architecture and threading model for our traffic generator application using multi-queue and double-buffer schemes.
- We implement our system on two multicore servers and compare our system with a stock Linux baseline and a traditional mmap-based file I/O approach.

The rest of the paper is organized as follows. Section II introduces the system overview. In Section III we present our Linux Exokernel middleware. In Section IV we address our 10G NIC driver design and userspace stack processing. Next, we describe the traffic generator design in detail in Section V. We present experimental results and evaluation in Section VI. Lastly, we survey related work in Section VII and conclude in Section VIII.

II. System Overview

In this paper, we focus on three main components as shown in Figure 1 that include: 1) Linux Exokernel middleware and its kernel module; 2) userspace 10G NIC driver and protocol stack; and 3) traffic generation application.

A. Key Components

We leverage an open-source Linux Exokernel middleware in our design [2], which was originally developed by us two years ago. There is an Exokernel module sitting inside the Linux kernel. It communicates with its userspace counterpart via the *proc* filesystem. Essentially, the Exokernel middleware abstracts core OS functionalities (*e.g.*, memory management, interrupt handling, scheduling, PCI subsystem) and allows device drivers, protocol stacks, schedulers, and memory managers (*e.g.*, paging) to run as userspace processes.

We develop a customized 10G NIC driver along with a scalable protocol stack on top of Exokernel middleware. By bypassing Linux kernel, our userspace driver and stack can produce much better packet throughput and latency. The dash arrows in Figure 1 show the control flow between the NIC and the userspace driver and stack.

In addition, the traffic generation application sits on top of the system architecture. To access files on disk, it communicates with storage devices via the normal Linux in-kernel driver. To send traffic, it interacts with the NIC via our driver and stack.

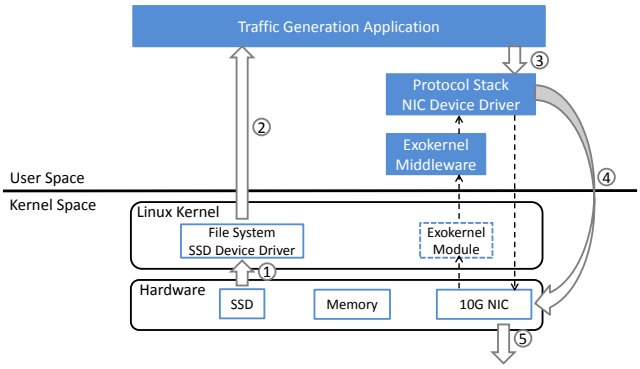


Fig. 1. The overview of system architecture.

B. Traffic Generation Workflow

The overall workflow for our traffic generator consists of five major steps as indicated in Figure 1. As an example, we assume the time series data files are stored in an NVMe SSD drive, and we use Intel X540 10G NIC to send out traffic.

- Step 1: The Linux in-kernel SSD driver and filesystem initialize the SSD device.
- Step 2: The file is read into the traffic generator process through a regular file I/O operation in read-only mode.
- Step 3: The traffic generator processes the time series records and sends them out based on timestamps by passing the data to the userspace stack and NIC driver.
- Step 4: The NIC driver configures the card to transfer the packets via DMA from host memory to device.
- Step 5: The record packets are sent out on the wire.

III. Linux Exokernel

The concept of an exokernel Operating System was originally developed by MIT [20], [21] with similar ideas being developed at the University of Glasgow and Microsoft Research in their Nemesis project [28]. The basic concept is to eliminate the role of a traditional kernel and allow applications (sometimes referred to as library operating systems) to directly interact with the hardware. This low-level hardware access allows programmers to implement custom abstractions, and omit unnecessary ones to improve performance.

Based on this idea, we developed an open-source eXokernel Development Kit (XDK) [2]. The implementation of XDK is not a true exokernel approach according to the strict MIT definition, but it does provide an exokernel-like capability on top of Linux. It allows developers to write applications that directly interact with the hardware and bypass the Linux kernel altogether primarily as a means to improve performance and avoid building new functionality in the kernel which could be easily implemented in user space. The Linux UIO framework [6] and the Intel DPDK architectures [4] are comparable to the XDK. The main differentiation is that XDK is unified into a single kernel module and also provides resource access control across multiple applications.

Figure 2 illustrates our Exokernel framework and shows two examples: 1) block device driver and file system; and 2) NIC driver and protocol stacks. Without loss of generality, we concentrate on the latter as it is more relevant to us.

The XDK implementation includes a dynamically loadable kernel module. It communicates with the kernel and can take control of physical resources, such as NICs, interrupt handling, thread scheduling and memory pages. The module provides low-level APIs to its userspace counterpart via the *proc* filesystem, as shown in solid blue blocks in Figure 2. Essentially, Exokernel middleware abstracts the hardware and OS core functionalities needed by NIC drivers, such as memory allocation and mapping, interrupt configuration and PCI address space access, allowing NIC drivers and protocol stacks to run in user space.

Compared to conventional OS where NIC drivers and network protocols are both kernel components, the Exokernel design bypasses the slow kernel path and permits user applications to interact with NIC directly (*e.g.*, avoid unnecessary memory copy and context switches). Additionally, we have full freedom to implement an efficient and scalable network subsystem on top of it. Thus, the Exokernel serves as a solid foundation to our traffic generator design.

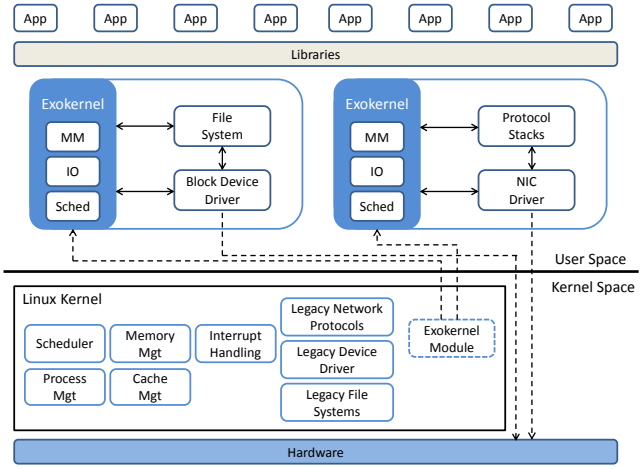


Fig. 2. The Linux Exokernel framework.

IV. Network Subsystem

A. Customized Scalable 10G NIC Driver

We target Intel X540 10G NIC to develop a scalable driver solution. Figure 3 illustrates the NIC driver architecture. From the bottom up, the flow director is essentially a load distributor that redirects incoming packets to different receive queues based on certain rules. At the next layer, we distribute receive/-transmit (RX/TX) queues to different cores, and have RX/TX threads running on each core to manage queues assigned to them. We also bind each Message Signaled Interrupts (MSI-X) vector to a specific core, so that interrupt handling can take place in parallel on a per-core basis. Figure 3 shows one configuration of the mapping between core, thread, queue and vector. Our design is motivated by the distributed resource partitioning coupled with data localization for performance optimization [37] [38].

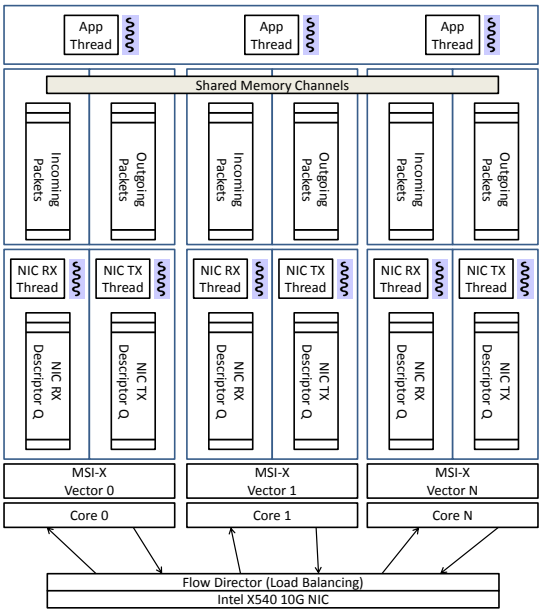


Fig. 3. NIC driver architecture overview.

Moving upwards, the shared memory channels sit between the NIC driver and application threads. The lock-free channels provide an efficient means for the driver and application to exchange data. There are two channels per core, one for incoming packets and the other for outgoing packets. We use shared memory for data transfer so that no extra memory copy is needed. The top layer represents a multithreaded application.

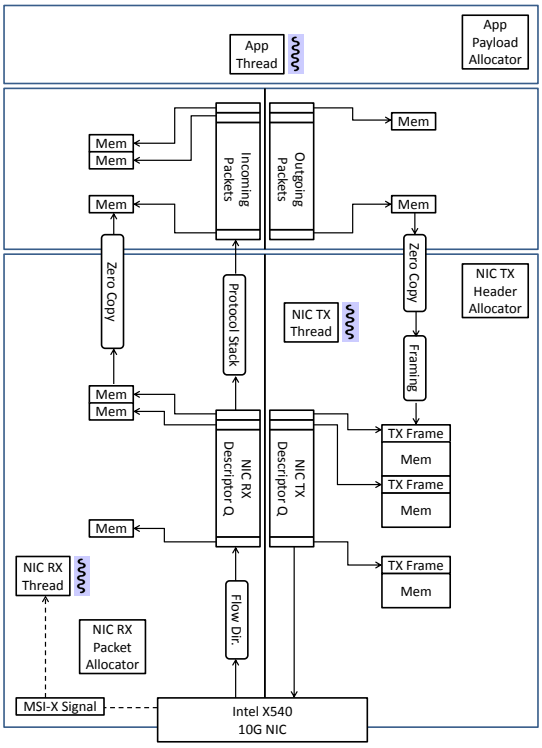


Fig. 4. Detailed NIC driver operation.

Figure 4 illustrates packet reception and transmission operation of our NIC driver on a single core (a single column in Figure 3). We focus on the transmit path in the interest of our traffic generator. The application thread first allocates

the payload memory. Then it pushes the data down to the shared memory channel, so that a TX thread can get it without copy overhead. Once the TX thread obtains the payload data, it will go through the necessary stack processing (*e.g.*, IP fragmentation and network header allocation) and prepare the TX descriptors. Packet header and payload can come from separate memory buffers to avoid copying. In our design, each outgoing packet will use two descriptors, one pointing to the payload memory, the other pointing to the network header.

B. Zero-copy User-level Stack

Unlike many other systems that employ zero-copy memory strategies but still incur copies within the kernel (*e.g.*, [14], [16], [17], [22] and [30]), our solution provides true zero-copy by customizing the IP protocol stack and using shared memory between the NIC driver and the application [38]. In the context of our system, zero-copy means the following:

- Network packets are DMA-transferred directly from the NIC device to user-space memory, and vice versa.
- Incoming packets are not assembled (defragmented) into the larger IP frames. Instead, they are maintained as a linked-list of packet buffers.
- Outgoing packets are dynamically composed by payload and network headers. There is no need to copy the complete packet in contiguous memory.

Figure 5 highlights the zero-copy data communication in our network subsystem between three logical layers identified by their respective functional roles, namely the network I/O layer, the stack processing layer and the application layer. In the following, we focus on the stack processing layer using UDP as an example.

The stack processing layer performs four major functions: 1) *IP Reassembly*, 2) *IP Fragmentation*, 3) *Header Formation*, and 4) *Ethernet Frame Construction*. First, *IP Reassembly* reassembles fragmented IP packets into a UDP packet. Different from conventional approaches, our *IP Reassembly* is realized by chaining fragmented IP packets into a linked list through manipulation of packet buffer pointers instead of memory copy. Second, *IP Fragmentation* breaks a UDP packet into multiple IP packets. In our solution, this is done by partitioning the payload data and allocating a separate network header for each logically fragmented IP packet organized by a linked list. Third, *Header Formation* prepares the network headers including Ethernet header, IP header and UDP header for packet transmission. Fourth, *Ethernet Frame Construction* prepares the entire Ethernet frame via packet composition.

We focus on the transmit path (steps 3 and 4 of Figure 5) as it is most relevant to our traffic generator. When the application passes down a payload, this layer takes the payload pointer and length as input. Based on the payload length, we know how many IP fragments are needed to send the whole payload. The stack processing layer will partition the payload by obtaining appropriate start and end addresses and form them into a linked list (see IP packets in Figure 5). At the same time, it will allocate new network header buffer for each IP fragment. This pointer traversal procedure is equivalent to the more expensive IP fragmentation process in traditional in-kernel stack. Next, for each IP packet in the list, this layer prepares its network header (including Ethernet header, IP header, and UDP header). In the end, the address and size of network header buffer and payload fragment are passed down to the network I/O layer for transmission.

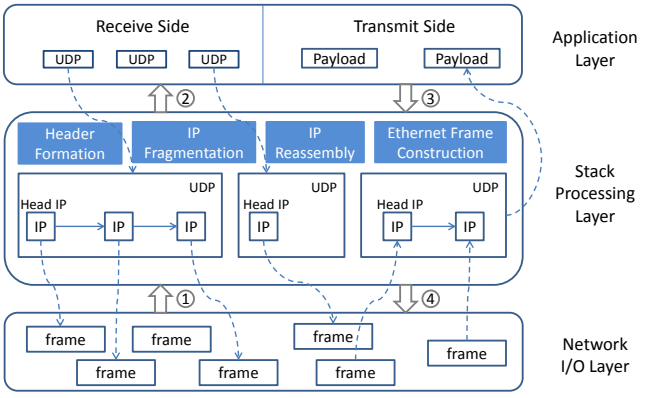


Fig. 5. Zero-copy data communication in stack processing.

V. Traffic Generator Design

In this section, we address three aspects of our traffic generator design: the archived data file format, the software architecture and threading model, and the key simulation features.

A. Archive File Format

As opposed to synthetic traffic generators that generate traffic based on parameterized models, ours reproduces archived data in a strictly time-conforming manner. To facilitate fast record retrieval and efficient timestamp parsing, we use a binary file format specifically designed for this purpose (see Figure 6). Each data file consists of an arbitrary number of records in a binary representation. Each record consists of four fields:

- Length (2 bytes): This is the total length of the record. This field is used to locate the next record's starting position. We choose 2 bytes since it can represent the maximum IP packet size.
- Timestamp (12 bytes): This is the encoded ISO 8601 time format (*i.e.*, complete date plus hours, minutes, seconds and a decimal fraction of a second). The finest time granularity is 1μs.
- Device ID (16 bytes): This is the device ID field, which can be used to enable per-device simulation control, such as dynamically adjusting the number of devices simulated, arbitrary device and flow binding.
- Data (up to max IP packet size): This is the actual time series data payload. This field is completely opaque to the traffic generator, and will be blindly copied into the packet buffer for transmission.

We encode the ISO 8601 time format into a compressed binary representation using 12 bytes as shown in Table I. As specified in the ISO standard, the time format is defined as follows: **YYYY-MM-DDThh:mm:ss.sTZD**, with exact punctuation. Note that the first "T" appears literally in the string,

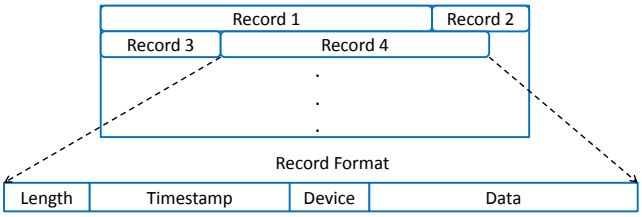


Fig. 6. Time series data input file format.

to indicate the beginning of the time element. Our encoding scheme is as follows:

- YYYY (4-digit year): 2 bytes (4 bits per digit).
- MM (2-digit month): 1 byte (4 bits per digit).
- DD (2-digit date): 1 byte (4 bits per digit).
- hh (2-digit hour): 1 byte (4 bits per digit).
- mm (2-digit minute): 1 byte (4 bits per digit).
- ss (2-digit second): 1 byte (4 bits per digit).
- s (up to 6-digit μsec): 3 bytes (4 bits per digit).
- TZD (time zone designator, Z or +hh:mm or -hh:mm): 2 bytes (1 bit for plus/minus sign, 3 bits for the first h, and 4 bits for the remaining digits).

Original Timestamp	Encoded Timestamp
2015-07-01T19:20:30.45+01:00	0x20 0x15 0x07 0x01 0x19 0x20 0x30 0x45 0x00 0x00 0x01 0x00
2015-07-01T19:20:30.456789-01:00	0x20 0x15 0x07 0x01 0x19 0x20 0x30 0x45 0x67 0x89 0x81 0x00
2015-07-01T19:20:30.4567Z	0x20 0x15 0x07 0x01 0x19 0x20 0x30 0x45 0x67 0x00 0x00 0x00

TABLE I
BINARY ENCODING EXAMPLES OF TIMESTAMPS.

B. Software Architecture and Threading Models

Figure 7 shows our traffic generator architecture and threading model, which is motivated by multicore parallelism, lock-free data structures and flexible thread configuration. We use one `IO thread` to read data files and copy them into double-buffered memory. The number of double-buffers depends on the number of `worker threads`, which is configurable. Each `worker thread` sequentially fetches time series records from memory buffer(s) and pushes them into a multi-producer multi-consumer (MPMC) lock-free queue. In addition, `TX threads` serve as consumers for the MPMC queue. They pull records from the queue, process them and send them out via the NIC driver.

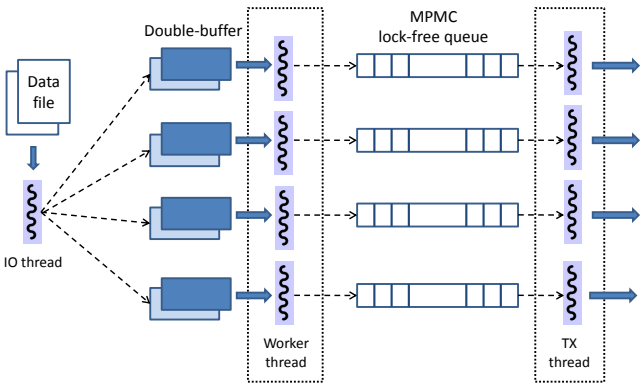


Fig. 7. Traffic generator architecture and threading model.

Next, we highlight the four main aspects of our design.

1) Double-buffer Management: To make sure the disk I/O does not become a performance bottleneck, we develop a double-buffer management scheme. When `IO thread` fills up a buffer, `worker threads` can access them immediately. In the meantime, `IO thread` can fill up the other buffer without waiting. Not only can this scheme effectively overlap disk I/O and memory read, it can also contribute to optimal disk throughput by using sequential reads.

By default, we pre-allocate 64MB for each buffer. `IO thread` often reads data from large files, so we open files with `O_RDONLY` and `O_LARGEFILE` flags. Repeatedly, `IO thread` finds an empty buffer and fills it up at once. In order to avoid partial records read, `IO thread` also scans the filled buffer and counts the actual number of records in the buffer. The file cursor is adjusted accordingly so that every time when `IO thread` reads the data file, it starts from the beginning of a new record. Consequently, when `worker thread` pulls records from the buffer, it knows exactly how many records have been buffered, which makes buffer synchronization between `IO thread` and `worker thread` simple. For better scalability, we give a double-buffer to each `worker thread` to eliminate contention among them.

2) MPMC Lock-free Queue: Our traffic generator leverages an efficient multi-producer multi-consumer (MPMC) lock-free queue from the Boost C++ library [1], which is essential for high concurrency and good scalability. `Worker threads` and `TX threads` act as producers and consumers, respectively. For each queue, we can have an arbitrary number of producers and consumers. As shown in Figure 7, we can configure more than one such queue in parallel, which further enhances the throughput and scalability.

With respect to the enqueued entry, it contains the following fields: record offset in the data file, record size, device ID, flow ID and buffer ID. In this way, each entry uniquely represents a time series record without copying the actual record. When `TX thread` dequeues an entry for timestamp processing, it is able to access the right record with low overhead. Only when `TX thread` is about to send the record does it copy the record to the packet buffer for transmission.

3) Flexible Threading Models: We designate three roles in our threading model: `IO thread` is responsible for disk I/O read and buffer write; `worker thread` prepares the records asynchronously by pushing them into the queue; and `TX thread` deals with timestamp processing and transmission. The minimum requirement is one thread for each role. Other than that, the number of actual threads per role is flexible and can be configured according to the underlying hardware as well as archived time series data characteristics for optimal performance.

Thanks to the scalable software architecture and the MPMC lock-free queue, it is trivial to modify thread roles in our traffic generator. For example, in our settings, one `IO thread` is enough to keep up the disk I/O, as we use an NVMe SSD with sequential read throughput up to 3GB/s. Thus, for the remaining threads, we can evenly distribute them into `worker threads` and `TX threads`, with the goal of achieving the

best throughput.

4) Timestamp Synchronization: As our traffic generator deals with real time series data, there are two challenges to be resolved with regard to timestamp synchronization: first, how to efficiently process timestamp and synchronize the host time with record timestamps; and second, how to synchronize the host time across multiple `TX threads` to keep time consistency for all records.

We tackle the first challenge by taking advantage of the Time Stamp Counter (TSC) for high-resolution time measurement. Lightweight `RDTSC` instruction is used to probe the current host TSC. When `TX thread` fetches the first record, it records the host TSC (C_h) and the record timestamp (T_0). These two values are kept as a reference point. For a future record with timestamp T_n, we first calculate the time difference between T_n and T_0. Based on the CPU frequency, we can derive the TSC difference between them, say C_d. Then, we can obtain the expected transmitting time for the new record by adding C_d to C_h. Lastly, before calling the NIC driver, we check the current host TSC C_n. If C_n is larger than C_d+C_h, we drop the record due to time violation; otherwise, we wait until C_n is equal to C_d+C_h and then send it out.

To tackle the second challenge, we maintain a global timestamp (T_g) for the very first record in the data file and per-thread host starting TSC (C_h) as reference point. In this way, each `TX thread` can reference to its own C_h when processing a record, which results in time consistency for all records across threads. More specifically, when the traffic generator starts, a thread barrier is used to prevent all but one `TX thread` from fetching records. Immediately after the first record is fetched, its timestamp is recorded in a global variable. At the same time, each `TX thread` probes its own host TSC and saves the value. By keeping the per-thread host starting TSC and the global record starting timestamp, we are able to achieve our objective in an efficient way.

C. Key Simulation Features

In this section, we present the key simulation features of our traffic generator other than the high-fidelity reproduction of archived time series data. We believe that these features will accommodate numerous use cases for IoT applications.

1) Probabilistic out-of-order simulation: To simulate the network unpredictability, we devise a probabilistic out-of-order simulation feature in our traffic generator. This feature adds randomness to the original time series data. We add another MPMC lock-free queue for out-of-order records.

We define three parameters for this feature:

- The out-of-order delay window (T_w): This parameter defines the maximum time window an out-of-order record can be sent out. If a record carries a timestamp T_0, for out-of-order simulation, it may be sent out at any random time between T_0 and $T_0 + T_w$. By varying T_w, we can simulate a wide range of network conditions in terms of packet arrival time.
- The number of out-of-order `TX threads`: This parameter specifies out of all `TX threads`, how many threads will also be getting records from out-of-order queue, processing the records and sending the records at appropriate times. The transmitting time is randomly generated but bound by an out-of-order delay window.
- The out-of-order probability: This parameter defines the probability of out-of-order records out of all data, based on which `worker threads` decide where to push the records, normal work queue or out-of-order queue.

2) Multi-flow simulation: Archived time series data may come from a single source or multiple sources. To allow flexibility in simulating both scenarios, we add multi-flow simulation in our traffic generator. Basically, this feature is enabled by mapping device ID to flow ID (`src port`, `src IP`, `dst port`, `dst IP`, `proto`). During the *Header Formation* step in our customized stack processing, we fill out the header information with appropriate flow ID. From the host's point of view, this is comparable to Linux's raw socket interface. In addition, with multi-flow simulation we can dynamically change the number of simulated flows, apply arbitrary filtering and keep track of per-flow statistics.

3) Packets with varying interval simulation: By default, our traffic generator generates time-conforming time series data. However, chances are that developers want to have the freedom to vary the traffic rate. For instance, to blast traffic with zero interval, we can test the application performance in handling high velocity traffic. For system testing or diagnosis, we may want to slow down or speed up the real traffic rate by some factor. Consequently, in addition to the normal time-conforming mode, we define a *tight_loop* mode and a *fast_forward* mode. In the *tight_loop* mode, we use a parameter (*i.e.*, *tight_loop_interval*) to adjust the packet delay arbitrarily, which means the record timestamp is ignored and the record transmitting time is only guided by *tight_loop_interval*. In the *fast_forward* mode, we use a parameter (*i.e.*, *fast_forward_factor*) to control the traffic rate change factor based on the original record timestamps (acceleration or deceleration).

4) Drifted time window simulation: This is a unique feature in time series data generation, which defines a relaxation window for each record. Assume the drifted time window is configured to be T_d, then for any record with timestamp T_0 the latest time it can be sent out will be $T_0 + T_d$. Thus, after a record is fetched and processed (right before its final sending call), we check the current time and compare it with the record's timestamp T_0. If the current time is no later than T_0+T_d, we send the record out; otherwise, we drop the packet. Therefore, the larger the value of T_d, the less fidelity the time series data has, and the less packet drop rate. If T_d is equal to 0, all packets will be sent out at the exact time according to the timestamp. As a result, we can rely on T_d and packet drop rate to measure the latency and jitter of our traffic generator in processing time series data.

VI. Experiments and Evaluation

A. Experiment Setup

We run our experiments on two platforms. The first one is a quad-core Intel workstation (one Intel Xeon E5-1620 CPU, 3.6GHz, 8GB memory, 500GB HDD), and the second one is a 20-core Intel server (two Intel Xeon E5-2670v2

CPUs, 2.5GHz, 64GB memory, 4TB HDD). Both platforms are equipped with a dual-port Intel X540 10G card [5] and a 400GB Samsung XS1715 NVMe SSD [9]. The NVMe SSD has a sustained read latency of 90μs and sequential read throughput up to 3GB/s. In the experiments, the target machine is connected to another receiving machine via point-to-point 10G connection. We use Ubuntu 14.04 with Linux kernel version 3.13.0.

To carry out a comprehensive performance study, the data sources we use include both real archived data and synthetically generated files. For the real data, we choose a seismic data set [11] that contains a period of 24 hours of continuous sampling from 1399 sensors. The total number of records is about 2.8 billion and each record is 28 bytes. For the synthetic data, we generate different source files with varying number of records, sampling frequency, device number, time duration and record size.

We refer to our traffic generator as `trafgen-exo` in this section for simplicity. To compare with other schemes, we implement the following two alternatives with varying threading models.

- `trafgen-stock`: This scheme deploys our traffic generation application directly to stock Linux using Intel's ixgbe driver and raw sockets. Transmit Packet Steering (XPS) is configured to assign each TX queue exclusively to a core, so that there is no contention when transmitting packets. NIC IRQs are also affinitized to their corresponding cores and `irqbalance` is disabled to prevent OS from routing IRQs to different cores.
- `trafgen-mmap`: This scheme replaces the double-buffer design in `trafgen-exo` with traditional mmap-based file I/O (we use Boost library's implementation [1]). In this scheme, `worker threads` are removed altogether, since `IO thread` can directly write into MPMC lock-free queue. In addition, multiple MPMC queues are not an option anymore considering there is only one `IO thread`. Thus, all `TX threads` access the same queue to get records.

Next, we present our extensive experimental results and show that our traffic generator can significantly outperform the other alternatives in terms of throughput, scalability and time violations. For throughput and scalability performance, we enable *tight_loop* mode with zero interval to observe the peak performance. In addition, for all runs, the number of threads is equal to the number of cores on both platforms.

B. Single TX Thread Throughput Performance

In this section, we compare the throughput performance of `trafgen-exo` and `trafgen-stock` with single `TX thread` for both SSD and HDD on the quad-core workstation. Figure 8 shows the result for different record sizes (ranging from 4 bytes to 1472 bytes, which represents the largest payload for a single Ethernet frame). Data is collected after sending 100 million records. From this figure, we clearly see that `trafgen-exo` outperforms `trafgen-stock` in nearly all scenarios, especially for small record sizes on SSD. In both 4-byte and 28-byte cases on SSD, `trafgen-exo` produces 2.4 million-packets-per-seconds (MPPS), whereas `trafgen-stock` only makes 0.98 MPPS. In addition, although for large records `trafgen-stock` is comparable to `trafgen-exo` due to much longer transmission delay, we still observe a 1.9X improvement in the case of 512-byte records on SSD, with 0.9 MPPS for `trafgen-stock` and 1.7 MPPS for `trafgen-exo`. In summary, Figure 8 proves our traffic generator benefits significantly from a more efficient network subsystem by using Exokernel and customized device driver and stack processing.

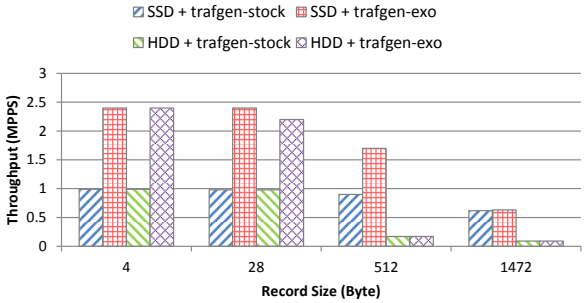


Fig. 8. Throughput comparison with trafgen-stock for both SSD and HDD.

C. Double-buffer and Mmap Comparison

In this section, we compare the throughput performance of `trafgen-exo` and `trafgen-mmap` for both SSD and HDD on the quad-core workstation. We use the same settings as previous experiment. Figure 9 shows the result for different record sizes. We come to the following two conclusions: 1) For single `TX thread`, both schemes have comparable performance, as Boost library's memory-mapped file implementation includes memory caching (prefetching), which is beneficial to sequential read workload as in our case. 2) However, `trafgen-mmap` suffers from scalability issue, as can be seen in this figure when we increase the number of `TX threads` to 2 and 3. In fact, we do not observe any performance gain when more threads are used. This disadvantage can be explained by the fact that `trafgen-mmap` only has one MPMC queue. As a result, if single `TX thread` already saturates the queue, extra threads will simply fail to scale up.

D. Single NIC Scalability Performance

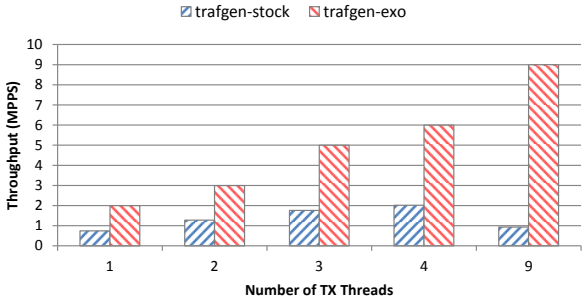
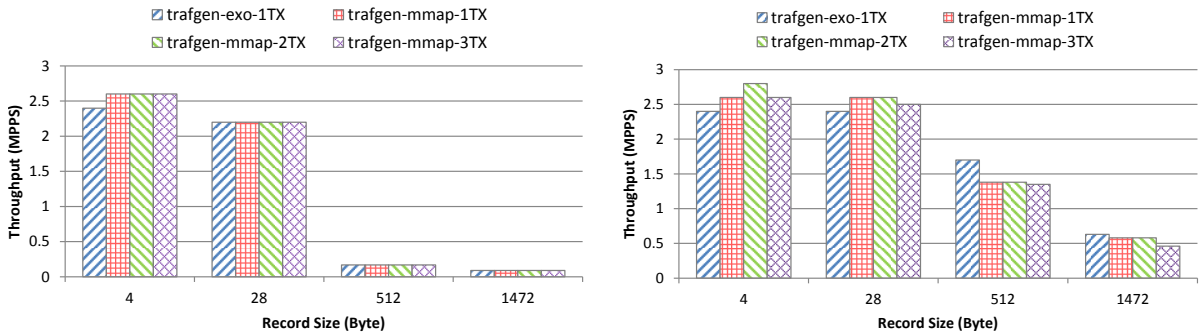


Fig. 10. Single NIC scalability comparison with trafgen-stock for SSD.

In this section, we compare the scalability performance of `trafgen-exo` and `trafgen-stock` with varying number of `TX threads` for SSD on the 20-core server. We use the threading model with equal number of `worker threads` and `TX threads`. Figure 10 shows the result for seismic data (28-byte records). Data is collected after sending 1 billion



(a) HDD (b) SSD

Fig. 9. Throughput comparison with *mmap* for both SSD and HDD.

records. We observe that `trafgen-exo` offers much better scalability as the number of threads increases, changing from 2 MPPS for 1 thread to 9 MPPS for 9 threads. On the contrary, `trafgen-stock` suffers from poor scalability due to Linux kernel overhead and contention in both stack and driver. For example, in the case of 9 `TX threads`, our scheme exhibits 3.6X improvement in scalability with a 10X performance difference. Thanks to our efficient and scalable design in both network subsystem and traffic generation application, `trafgen-exo` is able to scale its throughput performance much better than `trafgen-stock`.

E. Multi-NIC Scalability Performance

In this section, we extend our scalability comparison from single NIC to multi-NIC scenario. All experimental settings are the same as previous one, except that we test two 10G ports by launching two traffic generator processes and compare different record sizes for `trafgen-exo` and `trafgen-stock`, respectively. As can be seen in Figure 11(b), when we double the number of NICs, `trafgen-exo` perfectly scales up the throughput performance by a factor of 2, regardless of the record size and the number of `TX threads`. A peak throughput of 13.6 MPPS has been observed for `2NIC-4TX` case with 4-byte records. However, in comparison, `trafgen-stock` not only shows an average of 27% degradation for 4-byte records, but also it exhibits substantial negative impact when adding another NIC for larger record sizes, as shown in Figure 11(a). The throughput drops by 1% for 28-byte records, 31% for 512-byte records and 35% for 1472-byte records. Again, we attribute our superior scalability performance to the proposed whole-system optimization approach in designing `trafgen-exo`.

F. Time Violation Comparison

In this section, we compare the number of time violations that `trafgen-exo`, `trafgen-mmap` and `trafgen-stock` experience in terms of dropped packets for both HDD and SSD on the quad-core workstation. Packet drop due to time violation can result from both long latency and large jitter in handling high frequency time series data. We compare their respective performance using both synthetic and real data.

1) Synthetic data: We choose four synthetic data files with different number of records, devices and time durations as shown in Table II. Each record is 70 bytes in this study. All time series records are evenly distributed within the time range. For the first three data sets, we generate one record per 1μs. For the last data set, we generate two records per 1μs. We compare the performance with different values of drifted time window. As can be seen, `trafgen-exo` has the fewest dropped packets across all scenarios but one. In the best case, `trafgen-exo` has zero packet drop when the drifted time window is between 10-25μs, whereas `trafgen-mmap` and `trafgen-stock` drop up to 1.3% and 13.1% of the packets respectively with the same setting. In addition, even when the drifted time window is set to 0 in the most stringent case, `trafgen-exo` is able keep the drop rate below 0.5%, which is 8.3X better than `trafgen-mmap` and 46.2X better than `trafgen-stock`.

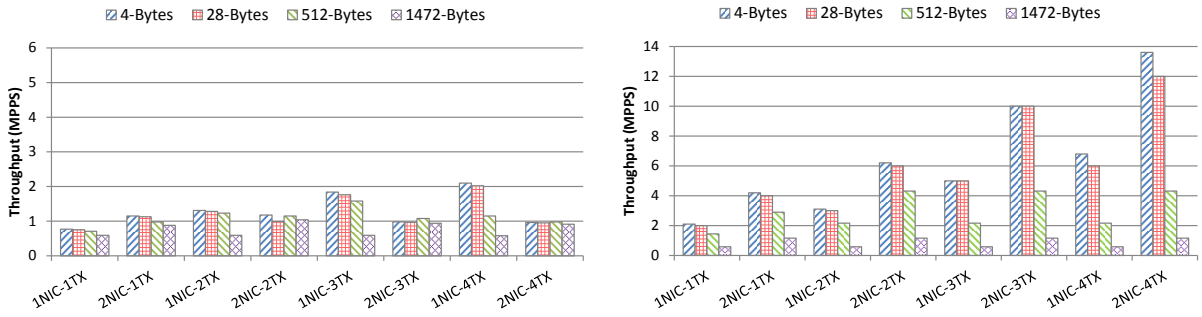
#Rec	#Dev	Time (sec)	Drift (μs)	#Dropped Packets (HDD / SSD)		
				trafgen-mmap	trafgen-exo	trafgen-stock
1M	1	1	0	11.7K / 15.7K	463 / 482	953 / 479
			5	10.4K / 14.6K	40 / 32	105 / 75
			10	7.7K / 9.5K	0 / 0	27 / 15
5M	1	5	0	16.9K / 15.3K	2.8K / 2.7K	4.2K / 3.7K
			10	15.9K / 6.3K	173 / 128	337 / 266
			20	14.7K / 2.9K	0 / 0	43 / 48
5M	100	5	0	21.7K / 7.9K	2.6K / 2.7K	15.1K / 8.8K
			10	16.3K / 7.1K	136 / 151	714 / 517
			20	11.9K / 5.1K	0 / 0	46 / 41
5M	2	2.5	0	217K / 173K	26K / 12K	1.2M / 929K
			10	211K / 160K	622 / 732	886K / 702K
			20	158K / 113K	74 / 104	811K / 637K
			25	65.1K / 46.3K	0 / 0	653K / 602K

TABLE II
TIME VIOLATION PERFORMANCE COMPARISON WITH SYNTHETIC DATA.

#Rec	#Dev	Time Duration	Drift (μs)	#Dropped Packets (SSD)	
				trafgen-exo	trafgen-stock
1M	1399	00:00:30	100	118	0.1M
5M	1399	00:02:33	100	124	0.6M
10M	1399	00:05:06	100	117	1.3M
50M	1399	00:25:33	100	122	6.3M
100M	1399	00:51:06	100	125	12.6M
2.8B	1399	23:54:43	100	173	347M

TABLE III
TIME VIOLATION PERFORMANCE COMPARISON WITH SEISMIC DATA.

2) Real seismic data: Table III shows the time violation performance for `trafgen-exo` and `trafgen-stock` using real seismic data. As the complete data file contains a period of 24 hours of records, we show 6 representative results with different time durations ranging from 30 seconds to 24 hours. As this data set contains 1399 sensors that can generate records at the same timestamp, we configure the drifted time window to be 100μs in all runs. From the numbers listed in this table we observe that `trafgen-exo`



(a) Trafgen-stock with multi-NIC for SSD.

(b) Trafgen-exo with multi-NIC for SSD.

Fig. 11. Multi-NIC scalability comparison with trafgen-stock for SSD.

outperforms `trafgen-stock` by a large margin. We see less than 200 dropped packets for `trafgen-exo` in all runs, even when we sent out the complete 2.8 billion records. However, `trafgen-stock` drops between 0.1 million and 347 million packets, which accounts for more than 10% of the total sent records in every single run.

VII. Related Work

A. Linux User-level Stack

In contrast to in-kernel stack, there are many prior works advocating the advantage of userspace stack, such as [10], [18], [19], [23], [24], [26], [27], [31] [32], [33] and [35]. Common arguments for implementing network protocols in user space include increased flexibility and customization, easier maintenance and debugging, and the possibility of application-specific optimizations offering better performance [18] [19] [23] [24] [35]. Arsenic [33] is a Linux 2.3.29-based userspace TCP implementation aimed at giving userspace applications better control for managing bandwidth on a specialized gigabit network interface. Arsenic enables zero-copy transfers and connection-specific buffers for the purposes of low overhead and QoS isolation. Based on Arsenic, Daytona [32] provides a very general userspace TCP stack that works with arbitrary network interfaces, with no kernel dependencies. However, the primary goal of Daytona is not performance-oriented. Two recent works include mTCP [26] and Arrakis [31]. First, mTCP [26] focuses on building a user-level TCP stack that provides high scalability on multicore systems. It leverages high-performance packet I/O libraries that allow applications to directly access the packets. Second, Arrakis [31] is a new OS designed to remove the kernel from the I/O data path without compromising process isolation. They use device hardware to deliver I/O directly to a customized user-level library (*e.g.*, network stack). In addition, to reduce data center application latency, the Chronos work [27] moves request handling out of the kernel to userspace by using zero-copy, kernel-bypass network APIs provided by several vendors with commodity server NICs such as [10].

Our scheme has both driver and stack in user space. Compared to aforementioned systems, we strengthen in scalable driver/stack design for multicore architecture, as well as "absolute" zero-copy throughout the packet's lifetime in the network subsystem.

B. Existing Zero-copy Techniques

1) User/Kernel Shared Memory: Typical examples include FBufs [17] and IO-Lite [30]. Those proposed techniques rely on shared memory semantics between the user and kernel address space and permit the use of DMA for moving data between the shared memory and network interface. The NIC drivers can also be built with per-process buffer pools that are pre-mapped in both the user and kernel spaces.

2) User/Kernel Page Remapping: Previous work belonging to this category include Copy on Write [16], Copy Emulation [14] and Trapeze [22]. These implementations re-map memory pages between user and kernel space by editing the MMU table and perform copies only when needed. They can also benefit from DMA to transfer frames between kernel buffers and the network interface.

3) User Accessible Interface Memory: LPC lwip buffer management [7], NTZC project [8], Intel's DPDK [4] and KV-Cache [38] enable user accessible interface memory. The network interface memory is accessible and pre-mapped into user and kernel address space. After incoming packets are stored in pre-allocated memory, their starting address is passed to applications directly. Thus, application can access the packet without memory copy. Same technique is also used when applications send a packet. There is no data copy at all for both incoming and outgoing packets.

The first two approaches mentioned above can effectively reduce the copy between kernel space and user space. However, they fail to remove the last copy due to packet fragmentation/defragmentation inside the kernel stack. In addition, page remapping schemes also require strict page alignment for packet buffers. The last approach is true zero-copy, but the first three works exclude protocol stacks and the last one (KV-Cache) is built for a microkernel system.

C. Network Traffic Generation

Much work in this area focuses on traffic modeling in various scenarios and aims at generating synthetic traffic based on realistic models. In [12], the authors study the time series models for Internet traffic based on real campus network traces. They argue that synthetic traffic generation based on their models is of great importance to simulation studies of resource management and sensitivity study of parameter estimates. Sommers *et al.* develop Harpoon [34], a new tool for generating representative IP traffic based on TCP and UDP flows using data collected from a NetFlow trace. Harpoon could be useful as a tool for providing network hardware designers and network operators insight into how systems might behave under realistic traffic conditions. In a similar work, [15] presents new source-level models for aggregated HTTP traffic and a design for their integration with the TCP

transport layer. The "connection-based" model derives from a large-scale empirical study of web traffic and is implemented in the *ns* network simulator. In addition, the authors in [36] propose a tool called Swing, which uses a comprehensive model to generate live packet traces by matching user and application characteristics on commodity OS subject to the communication characteristics of an emulated network. They claim that Swing will enable the evaluation of a variety of higher-level application studies, such as bandwidth/capacity estimation tools and dynamically reconfiguring overlays, subject to realistic levels of background traffic and network variability. Benson *et al*. [13] present a preliminary empirical study of end-to-end traffic patterns in data center networks that can inform and help evaluate research and operational approaches. They apply their framework to design network-level traffic generators for data centers. Lastly, [37] describes a workload generator system called KV-Blaster, which is based on a highly-optimized microkernel system and can generate requests for key-value store/cache system with throughput up to 4 MPPS rate. In KV-Blaster, it also applies an efficient and scalable userspace NIC driver and stack similar to ours.

Compared to these work, our traffic generator largely differs in the following two aspects. First, we focus on time series data replay from archived data in an efficient and scalable way. Second, we leverage state-of-the-art hardware/software and adopt a whole-system optimization approach.

VIII. Conclusion

The emergence of big data and IoT have brought numerous new challenges to application developers, who have to deal with unprecedented data volume and rate in their system. In order to test and optimize such systems, a highly efficient and scalable time series traffic generator is not only necessary but also challenging. In this paper, we adopt a holistic approach to tackle this problem. Our contribution lies in the following: 1) we develop a traffic generator that generates time-conforming packets based on high-fidelity reproduction of archived time series data; 2) our system is based on an open-source Linux Exokernel middleware and a userspace 10G NIC driver; 3) we take advantage of "absolute" zero-copy in stack processing; and 4) we propose an efficient and scalable software architecture and threading model. Experiments show that our traffic generator outperforms other alternatives in terms of throughput (10X), scalability (3.6X) and time violations (46.2X).

References

[1] Boost c++ libraries. http://www.boost.org.

[2] Exokernel development kit. https://github.com/dwaddington/xdk.

[3] Gartner research. http://www.gartner.com/technology/research/internet-of-things/.

[4] Intel data plane development kit (dpdk). http://dpdk.org.

[5] Intel ethernet controller x540 datasheet. http://www.intel.com/content/www/us/en/network-adapters/10-gigabit-network-adapters/ethernet-x540-datasheet.html.

[6] Linux uio framework. https://www.kernel.org/doc/htmldocs/uio-howto/about.html.

[7] Lpc lwip buffer management. http://www.lpcware.com/content/project/lightweight-ip-lwip-networking-stack/lwip-buffer-management.

[8] Ntzc project. http://code.google.com/p/ntzc/.

[9] Samsung xs1715 nvme ssd. http://www.samsung.com/global/business/semiconductor/file/product/xs1715_prodoverview_2014_1.pdf.

[10] Solarflare openonload. http://www.openonload.org.

[11] Usgs real-time seismic data. http://earthquake.usgs.gov/data/?source=sitenav.

[12] S. Basu, A. Mukherjee, and S. M. Klivansky. Time series models for internet traffic. *Tech Report GIT-CC-95-27. Georgia Institute of Technology*, 1995.

[13] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proc. of SIGCOMM CCR '10*, 2010.

[14] J. Brustoloni and P. Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *Proc. of INFOCOM '97*, 1997.

[15] J. Cao, W. S. Cleveland, Y. Gao, and F. Kevin Jeffay. Stochastic models for generating synthetic http source traffic. In *Proc. of INFOCOM '04*, 2004.

[16] H. J. Chu. Zero-copy tcp in solaris. In *Proc. of USENIX ATC '96*, 1996.

[17] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of SOSP '93*, 1993.

[18] A. Edwards and S. Muir. Experiences implementing a high-performance tcp in user-space. In *Proc. of SIGCOMM '95*, 1995.

[19] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proc. of USIT '01*, 2001.

[20] D. Engler and M. Kaashoek. Exterminate all operating system abstractions. In *Proc. of HotOS '95*, 1995.

[21] D. R. Engler. The exokernel operating system architecture. *MIT Ph.D. Thesis*, 1998.

[22] A. Gallatin, J. Chase, and K. Yochum. Trapeze/ip: Tcp/ip at near-gigabit speeds. In *Proc. of ATC '99*, 1999.

[23] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM TOCS*, 2002.

[24] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying safe user-level network services with ictcp. In *Proc. of OSDI '04*, 2004.

[25] S. Jankowski, J. Covello, H. Bellini, J. Ritchie, and D. Costa. The internet of things: Making sense of the next mega-trend. *Goldman Sachs Equity Research*, 2014.

[26] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *Proc. of NSDI '14*, 2014.

[27] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proc. of SOCC '12*, 2012.

[28] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 1996.

[29] A. Ltd. Releasing the value within the industrial internet of things. *Attunity Ltd White Paper*, 2014.

[30] V. Pai, P. Druschel, and W. Zwaenepoel. I/o lite: A unified i/o buffering and caching system. In *Proc. of OSDI '99*, 1999.

[31] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proc. of OSDI '14*, 2014.

[32] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona: A user-level tcp stack. *Tech Report*, 2002.

[33] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proc. of INFOCOM '01*, 2001.

[34] J. Sommers and P. Barford. Self-configuring network traffic generation. In *Proc. of IMC '04*, 2004.

[35] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proc. of SIGCOMM '93*, 1993.

[36] K. V. Vishwanath and A. Vahdat. Realistic and responsive network traffic generation. In *Proc. of SIGCOMM '06*, 2006.

[37] D. Waddington, J. Colmenares, J. Kuang, and R. Dorrigiv. A scalable high-performance in-memory key-value cache using a microkernel-based design. *Tech Report, http://dx.doi.org/10.13140/2.1.5084.6086*, 2014.

[38] D. Waddington, J. Colmenares, J. Kuang, and F. Song. KV-Cache: A scalable high-performance web-object cache for manycore. In *Proc. of UCC '13*, 2013.